# DESIGN AND POWER ESTIMATION OF BOOTH MULTIPLIER USING DIFFERENT ADDER ARCHITECTURES

A THESIS SUBMITTED IN PARTIAL FULLFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

**Bachelor of Technology**

**In**

**Electronics & Communication Engineering**

**By**


**BIKASH CHANDRA SAHOO (109EC0234)**

**SANJAY KUMAR SAMANT (109EC0240)**

**Department of Electronics & Communication Engineering**

**National Institute of Technology, Rourkela**


**2013**

# DESIGN AND POWER ESTIMATION OF BOOTH MULTIPLIER USING DIFFERENT ADDER ARCHITECTURES

A THESIS SUBMITTED IN PARTIAL FULLFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

**Bachelor of Technology**

**In**

**Electronics & Communication Engineering**

**By**

**BIKASH CHANDRA SAHOO (109EC0234)**

**SANJAY KUMAR SAMANT (109EC0240)**

**Under the guidance of**

**PROF. K. K. MAHAPATRA**

**Electronics & Communication Engineering**

**National Institute of Technology, Rourkela**

**Department of Electronics & Communication Engineering**

**National Institute of Technology, Rourkela**

**2013**

**National Institute of Technology**

**Rourkela**

# CERTIFICATE

This is to certify that the thesis entitled "**DESIGN AND POWER ESTIMATION OF BOOTH MULTIPLIER USING DIFFERENT ADDER ARCHITECTURES**" submitted by **Mr. Bikash Chandra Sahoo,** Roll No: **109EC0234** & **Mr. Sanjay Kumar Samant**, Roll No: **109EC0240**, Final year students of Electronics and communication Engineering, in partial fulfillments of the requirements for the award of Bachelor of Technology Degree in Electronics and Communication Engineering at National Institute of Technology, Rourkela is an authentic work carried out by them under my supervision and guidance.

To the best of my knowledge, the matter embodied in thesis has not been submitted to any other university/ Institute for the award of any degree or Diploma.

Date:

**Prof K. K. Mahapatra**

**Department of E.C.E**

**National Institute of Technology**

**Rourkela- 769008**

# ACKNOWLEDGEMENT

We would like to articulate our profound gratitude and indebtedness to our project guide Prof. Dr. K. K. Mahapatra who has always been a constant motivation and guiding factor throughout the project time in and out as well. It has been a great pleasure for us to get an opportunity to work under him and complete the project successfully.

We wish to extend our sincere thanks to Prof. Dr. S. Meher, Head of our Department, for approving our project work with great interest.

We would like to mention Mr. Sudeendra Kumar for his cooperation and constantly rendered assistance.

An undertaking of this nature could never have been attempted without our reference to and inspirations from the works of others whose details are mentioned in references section. We acknowledge our indebtedness to all of them. Last but not the least, our sincere thanks to all our friends who have patiently extended all sorts of help for accomplishing this undertaking.

Bikash Chandra Sahoo
Roll No: 109EC0234
Electronics & Comm. Engineering
NIT Rourkela

Sanjay Kumar Samant
Roll No: 109EC0240
Electronics & Comm. Engineering
NIT Rourkela

# CONTENTS

# ABSTRACT

Modern IC Technology focuses on the design of ICs considering more area optimization and low power techniques. Multiplication is a heavily used arithmetic operation that figures prominently in signal processing and scientific applications. Multiplication is a very hardware intensive subject and we as users are mostly concerned with getting low-power,smaller area and higher speed.The most important concern in classic multiplication, mostly realized by K-cycles of shifting and adding, is to speed up underlying multi-operand addition of partial products. In this project we will present the design of Booth Multiplier with different adder architectures like Ripple Carry Adder & Carry Look Ahead Adder. The time delay, area and power have been analyzed for different adders. Also multipliers have been designed for both radix-2 and radix-4. Results will show the variation of area, speed and power for different designs. Also the power estimation method gives the deeper insight into power calculation and analysis. An approach have been suggested for peak power estimation.

# LIST OF FIGURES



# LIST OF TABLES

# CHAPTER 1

## INTRODUCTION

- MOTIVATION
- MULTIPLIER DESIGN
- ANALYSIS TOOLS USED
- REASEARCH APPROACH

## 1.1 MOTIVATION

Day by day IC technology is getting more complex in terms of design and its performance analysis. A faster design with lower power consumption and smaller area is implicit to the modern electronic designs. Unceasing advancement in microelectronics design technology makes improved use of energy, encrypt data successfully, communicate information much more steadfastly, etc. Particularly, many of these technologies address low-power consumption to meet the requirements of various portable applications. In these application systems, a multiplier is a fundamental arithmetic unit and widely used in circuits, for which the multiplication process should be optimized properly. Multipliers generally have extended latency, huge area and consume substantial amount of power. Hence low-power multiplier design has become an important part in VLSI system design. Everyday new approaches are being developed to design low-power multipliers at technological, physical, circuit and logic levels. Since the multiplier is generally the slowest element in a system, the system's performance is determined by performance of the multiplier. Also multipliers are the most area consuming entity in a design. Therefore, optimizing speed and area of a multiplier is a major design issue nowadays. However, area and speed are usually conflicting constraints so that improving speed results in larger areas and vice-versa. Also area and power consumption of a circuit are linearly correlated. So a compromise has to be done in speed of the circuit for a greater improvement in reduction of area and power.

For implementing a digital multiplier a large variety of computer arithmetic algorithms could be used. Most techniques take into consideration generating a set of partial products, and then adding the partial products together once they have been shifted. In a multiplier to increase its speed, the number of partial product to be genrated should be reduced. A higher representation radix effectively indicates to fewer digits. Thus, a single-digit multiplication algorithm necessitates fewer cycles as we start moving to much higher radices, which automatically leads to a lesser number of partial products. Several algorithms have been developed for this purpose like Booth's Algorithm, Wallace Tree method etc. For the summation process several adder architectures are available viz. Ripple Carry Addition, Carry Look-ahead Addition, Carry Save Addition etc. But to reduce the power consumption the summation architecture of the multiplier should be carefully chosen.

### 1.2 LOW POWER MULTIPLIER DESIGN

Multiplication can be considered to consist of three basic steps: generation of partial product (PPG), partial products reduction (PPR), and finally at the end addition of carrypropagate(CPA).In general we have combinational and sequential multiplier implementations. Here we are taking into consideraion the combinational case only, because the scale of integration now has become huge enough to start accommodating parallel multiplier applications in digital VLSI circuits. Different multiplication algorithms vary in the approaches of generation and reduction of Partial Products and the addition process. In order to diminish the number of PPs involved and therefore lessen the area/delay of the circuit, one operand is usually recoded into high-radix digit sets. One of the most used and widespread radix-2n algorithm is the radix-4 which has a set of digits given by {-2,-1, 0, 1, 2} for PPG. For PPR, two choices exist which can be implemented: reduction by rows, which can be performed by taking into consideration an adder array and reduction by columns, which can be performed by taking into consideration a counter array .The closing process of addition necessitates a fast adder arrangement because it is on the critical path. In a few cases, concluding summation is deferred if it is valuable to keep redundant results from PPG to carry out further arithmetic operations.

### 1.3 PROGRAMMING LANGUAGE AND ANALYSIS TOOLS USED

To write program for the implementation of any digital circuit there are various languages available, called as Hardware Description Language e.g. Verilog, VHDL. For our design we have used VHDL (Very High Specific Integrated Circuit HDL) for programming. VHDL is one of the common techniques used in digital system emergent process. The technique is implemented in program using certain software which carries out simulation and examination of the designed system. The designer only needs to describe the digital circuit design in textual form which can remove without the effort to alter the hardware. VHDL is highly preferred because this technique has the ability to reduce cost and time, is easy to troubleshoot, portable, a lot of platforms software support the VHDL function and high references are available. We used XILINX 10.1 platform to write our programs. All the RTL simulations has been done using this software only. Also for delay report the synthesis tool embedded in Xilinx was used.

We used for Scirocco and VirSim, which are logic simulators, for the functionality simulation of our design. Also we used Synopsys Design Vision tool to estimate power of all our arithmetic circuits. Synopsys Design Vision is a logic synthesis tool. It takes HDL designs and synthesizes them to gate-level net-lists. Also it supports both Verilog and VHDL. It can synthesize generic gates or other design libraries. The tool exists inside a GUI and command line version. The GUI version is known as design vision and the command line version is referred as dc_shell-xg-t. For both area and power estimation we used Design Vision. The basic steps for analyzing a design are:

Analyze: This step start checking the design files for syntax.We can also save modules (Verilog) and entities (VHDL) in an intermediate format into a local folder.

Elaborate: We can build a design from the intermediate format files created in the previous Analyze step.

Compile: This is the synthesizing step, where we can map the design to a gate library or cell library.

Save: After compiling a design we can save the synthesized design into HDL or other formats. Synthesized designs are fundamental for creating ASICS or carrying out different simulations for timing and power.

After compilation using commands like *report_power* or *report_area* we can get power and area accordingly.

### 1.4 RESEARCH APPROACH

The elementary purpose of our project is to instrument the Booth's Algorithm for the design of a binary multiplier using different adder architectures and carry out power analysis at various levels. Also the delay, area and power optimization is to be taken care of. We chose to implement Booth's algorithm for our multiplier design because it reduces the number of partial products generated in a multiplication process and reduction in number of partial products results in a faster multiplication.

We already are familiar that the basic building block of a multiplier is the adder circuit. Therefore we turned our focus into The ADDERS first. We analyzed the occupied area and the delay in time consumed by different adders and discerned an appropriate relationship between time and area complexity of all the adders which we have taken under consideration. Then we turned our attention to the design and implementation of Multipliers. First of all we considered a Booth's Radix-2 multiplier and estimated its delay, area and power. Then a radix-4 multiplier was designed. A comparison was done between Radix-2 and Radix-4 algorithm. As radix-4 seemed more suitable for the design we carried out further analysis on radix-4 multiplier by using different adder architectures like RCA and CLA.

Then we turned our focus into the switching activity based power analysis of the Radix-4 Booth multiplier, and its power estimation. We did power estimation at RTL level using Synopsys Design Compiler.

# CHAPTER 2

## ADDERS

- ADDERS CLASSIFICATION
- RIPPLE CARRY ADDER
- CARRY LOOK-AHEAD ADDER
- ANALYSIS OF ADDERS
- DISCUSSION

## 2.1 ADDERS CLASSIFICATION

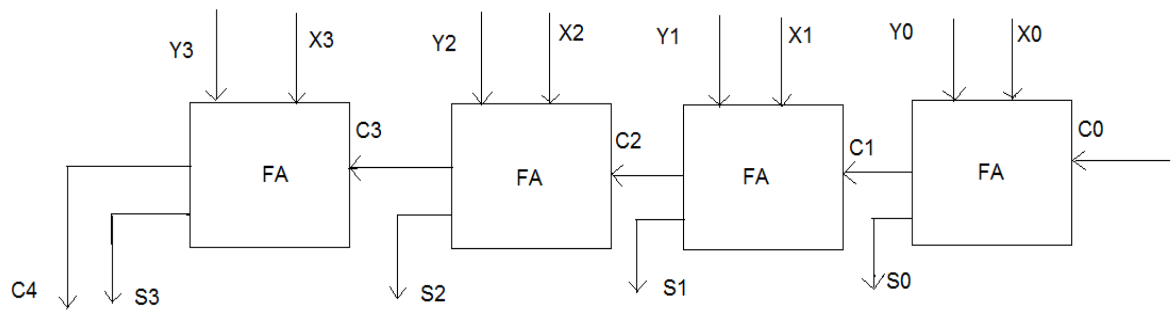
Addition is one of the most commonly used arithmetic operation in microprocessor, digital signal processor etc. It can also be used as a building block for synthesis of all other arithmetic operations. Therefore, as far as the efficient implementation of an arithmetic unit is concerned, the binary adder structure becomes a very critical hardware unit. In any book on computer arithmetic, we can observe that there occurs a large number of quite different circuit architectures pertaining to different performance characteristics. While adders can be constructed for a lot of numerical expressions like Binary-coded decimal or excess-3, the most frequently used adders operate numbers which are binary. In certain cases where two's complement is being used to represent negative numbers, it is trivial to convert an adder into an adder-subtractor.

Although many researches related to the binary adder structures have been carried out, the studies based on their comparative performance analysis are only quite few in number. In this project, assessments of the classified binary adder architectures are given. From the huge member of adders we have got, we implemented the VHDL (Hardware Description Language) code for Ripple-carry and Carry-look ahead adder to highlight the common performance properties belong to their classes. Throughout the next section, we provide you with a brief description of the studied adder architectures.

## 2.2 RIPPLE CARRY ADDERS (RCA)

This popular adder architecture, ripple carry adder consists of cascaded full adders as shown in figure.1.It is formed by cascading full adder blocks in series with one another. The output carry of one stage is fed directly to the input carry of the next stage. An ***N***-bit parallel adder requires ***N*** full adders.

**FIGURE 2.1**

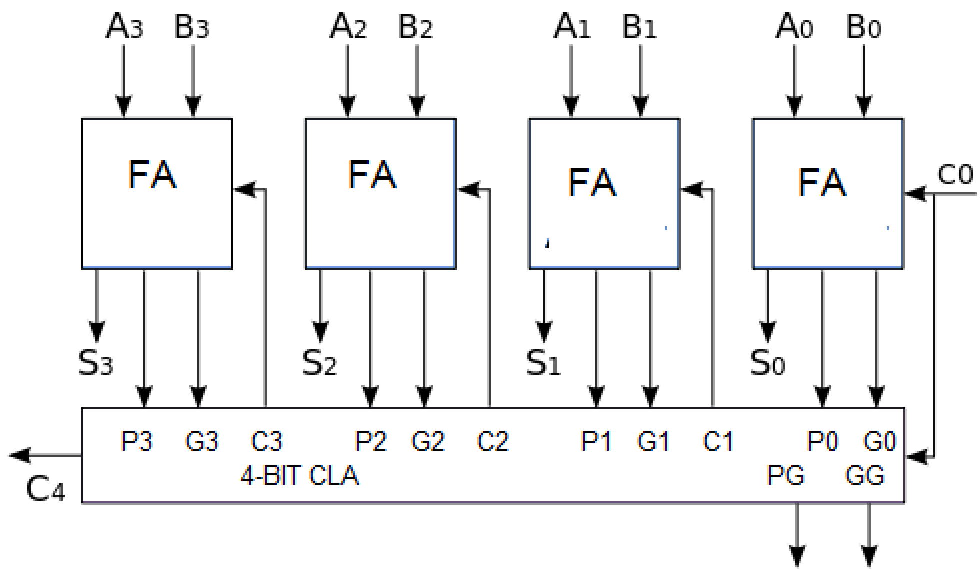



The given adder architecture is not very efficient when large number of bits are used. The gate delay can easily be calculated by inspecting the full adder circuit. We know that each full adder requires three levels of logic. Considering a 64-bit ripple-carry adder, we know that it has 64 full adders, so the critical path (worst case) delay is 3 (from input to carry in case of the first adder) + 63 * 2 (for carry propagation in the later adders) = 127 gate delays.

## 2.3 CARRY LOOK AHEAD ADDERS (CLA)

A Carry Look Ahead Adder has the ability to generate faster carries because of generation of carry bits in parallel by a supplementary circuit whenever inputs are changing. This technique extensively uses carry bypass logic to haste up the propagation of carry. In Carry look ahead logic the generation and propagation of carries takes place. For each bit in a binary sequence to be added, the Carry Look Ahead Logic determines whether that bit pair will generate a carry or propagate a carry. This allows the circuit to "pre-process" the two numbers being added to determine the carry ahead of time. After this, when the actual addition is performed, there will be no delay from waiting for the ripple carry effect (or time it takes for the carry from the first Full Adder to be passed down to the last Full Adder).

**FIGURE 2.2**




The mechanism for carry look-ahead summation can be describes as below:

First the Carry-generate and Carry-propagate vectors are evaluated.

$$P_i = A_i \oplus B_i$$

$$G_i = A_i B_i$$

$$S_i = C_i \oplus P_i$$

$$C_{i+1} = G_i + P_i C_i$$

*Gi* is known as the carry Generate signal because a carry ($C_{i+1}$) is generated whenever $G_i$ =1, regardless of the input carry ($C_i$).

$P_i$ is known as the carry propagate signal because whenever $P_i$ =1, the input carry is propagated to the output carry, i.e., $C_{i+1}$. = $C_i$

The Boolean expression for the carry outputs of various stages for a 4-bit block can be written as follows:

$C_1 = G_0 + P_0C_0$

$C_2 = G_1 + P_1C_1 = G_1 + P_1 (G_0 + P_0C_0) = G_1 + P_1G_0 + P_1P_0C_0$

$C_3 = G_2 + P_2C_2 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$

$C_4 = G_3 + P_3C_3 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0$

As the no of bits in the Carry Look Ahead adders increases, the complexity increases as the no. of gates in the expression $C_{i+1}$ increases. So practically it is not desirable to use the traditional CLA shown above because it increases the Space required and the power too.

## 2.4 ANALYSIS OF ADDERS

In our project we compared 2- different adders *Ripple Carry Adder and the Carry Look-Ahead Adder*. The basic purpose of our experiment was to know the time, area and power trade-offs between various adders which will give us a clear picture of which adder suits best in which type of situation during a design process. Hence below we present the practical comparisons of the two adders which were taken into consideration. There are a lot of adders present but we took into consideration only these two and our project is limited to these two adders. Limited to these two adders.

**TABLE 2.1: Power-Area Comparison for Different Adders**

| ***Name Of Architecture*** | ***Cell Internal Power (in uW)*** | ***Net Switching Power (in uW)*** | ***Total Dynamic Power(in uW)*** | ***Area (in $um^2$)*** |
|---|---|---|---|---|

| | | | | |
|---|---|---|---|---|
| *RCA-8 bit* | *13.4914* | *2.8803* | *16.3717* | *81.719999* |
| *RCA-16 bit* | *27.953* | *6.1735* | *34.1265* | *162.359999* |
| *RCA-64 bit* | *114.9175* | *25.9866* | *140.9041* | *646.199995* |
| *CLA-8 bit* | *6.0445* | *0.98704* | *7.0315* | *46.079999* |
| *CLA-16 bit* | *34.2506* | *10.7415* | *44.9921* | *253.799998* |
| *CLA-64 bit* | *137.4008* | *43.3389* | *180.7397* | *950.399992* |

**2. 5 DISCUSSION**

Above we have presented the estimated power and power of different types of adders with different sizes using Design Compiler by Synopsys.

For Ripple Carry Adder the time complexity is $O(n)$ *i.e.* the delay of the circuit caries linearly with the number of bits. Theoretical delay for the addition of $n$-bit data using RCA and CLA are $2n$ and $4log_2(n)$ respectively.

By looking at the above data it can be inferred that the total dynamic power i.e. summation of cell internal power and net switching power increases linearly with the number of bits for RCA architecture. But for CLA architecture it varies in a non-linear fashion, more like in an exponential way. Similarly the area also increases proportionally with number of bits for an RCA but it increases in an exponential way for a CLA architecture. The reason for linear increase in area and power for an RCA is that the number gates increases proportionally as the number of bits increases.

But for a CLA the carry look ahead logic circuit becomes more complex and larger with increment in number of bits. Later in this thesis we will give comparison about the multipliers designed using these two architectures.

# CHAPTER 3

## THE MULTIPLIERS

- **BASIC MULTIPLICATION ALGORITHM**
- **BOOTH'S ENCODING**
- **MODIFIED BOOTH'S ALGORITHM**

## 3.1 BASIC ALGORITHM FOR BINARY MULTIPLICATION

A Binary multiplier is an electronic device used in digital electronics or in a computer or other electronic devices to carry out multiplication of two numbers depicted in binary format. It is built using binary adders. The most basic technique involves generating a set of partial products, and then summing the partial products simultaneously. This process is similar to the method which is taught to lower classes' students in school for conducting long multiplication on base-10 integers, but has been modified here for application to a base-2 (binary) numeral system.

The rules for binary multiplication are stated as given:

1. If the multiplier digit is 1, the multiplicand is copied down and it gives the product.
2. If the multiplier digit is 0 then we get a product which is also 0.

For designing such a multiplier circuit we should have the circuitry to carry out or do the following four things:

1. It should be capable of recognizing whether a bit is 0 or 1.
2. It should be capable of shifting the left partial product.
3. It should be capable of adding all the partial-products to give the product as a sum of the partial products.
4. It should examine sign bits and if they are similar, the sign of the product will be a Positive representation and if the sign bits are opposite then the product will be negative. The sign bit of the product which has been stored with the above criteria should be displayed along with the product.

From the above discussion we can observe that it is not necessary to wait until all the partial products have been formed before carrying out the sum. In fact the addition of the partial products can be carried out as soon as a partial product is formed.

### 3.2 BOOTH'S ENCODING

Booth's encoding or Booth's multiplication algorithm[1] is a multiplication algorithm which can multiply two signed binary numbers in a two's complement notation. Booth's algorithm has the ability to perform fewer additions and subtractions in comparison to normal multiplication algorithm. It is an encoding process which can be used to minimize the no of partial products in a multiplication process. It is based upon the relation

$$2^n = 2^{n+1} - 2^n$$

Example:

0 0 1 1 1 1 1 1 0 0

+1 -1

+1 -1

+1 -1

+1 -1

+1 -1

+1 -1

0 +1 0 0 0 0 0 -1 0 0

Booth's algorithm examines consecutive bits of the N-bit multiplier Y in signed two's complement representation, which includes an implicit bit below the least significant bit, $y_{-1} = 0$. For each bit $y_i$, as i runs from 0 to N-1, the bits $y_i$ and $y_{i-1}$ are considered. When these two bits are equal, the product accumulator P stays unchanged. Where $y_i$ = 0 and $y_{i-1}$ = 1, the multiplicand times 2i is added to P; and where $y_i$ = 1 and $y_{i-1}$ = 0, the multiplicand times *2i* gets subtracted from P. The final value of P will be the signed product.

The representation of the multiplicand and product are not specified; typically, these are also in two's complement representation, like a multiplier, but any number system that supports addition and subtraction will work as well. The order of the steps is not determined. Generally, it proceeds from LSB to MSB, starting at i = 0; the multiplication by 2i is then replaced by incremental shifting of the P accumulator to the right between steps; low bits will be shifted out, and subsequent additions or subtractions can then be done just on the highest N bits of P. There are many variations and optimizations on these details.

The algorithm is often described as converting strings of 1's in the multiplier to a high-order +1 and a low-order –1 at the ends of the string. When the string runs through the MSB, there is no high-order +1, and the net effect is interpretation as a negative of the appropriate value.

**RADIX-2 ALGORITHM IMPLEMENTATION**

Let *x* be the number of bits of the multiplicand, and *y* be the number of bits of the multiplier:

- Draw a grid of three rows, each with columns for $x + y + 1$ bits. Label the lines respectively A (add), S (subtract), and P (product).
- In two's complement notation, fill the first *x* bits of each line with :
    - A: the multiplicand
    - S: negative of the multiplicand(2's complement format)

    - P: zeroes

- Fill the next *y* bits of each line with :

    - A: zeroes
    - S: zeroes
    - P: the multiplier

- Fill the last bit of each line with a zero.

For example consider the given two numbers: 3 and -4.

On carrying out the above instructions we would find the following values of A, S and P.

A = 0011 0000 0

S = 1101 0000 0

P = 0000 1100 0

- ➢ Now carry out both of these steps *y* times :

- .If the last two bits in the product are...

    - 00 or 11: do nothing.
    - 01: P = P + A. Ignore any overflow.
    - 10: P = P + S. Ignore any overflow.

- .Arithmetically shift the product right one position.

- Drop the first (we count from right to left when dealing with bits) bit from the product for the final result.
- Do both of these steps *y* times :

  - If the last two bits in the product are...
    - 00 or 11: do nothing.
    - 01: P = P + A. Ignore any overflow.
    - 10: P = P + S. Ignore any overflow.
  - Arithmetically shift the product right one position.

- Drop the first (we count from right to left when dealing with bits) bit from the product for the final result.

For Example: Find 3 × -4, with $x = 4$ and $y = 4$:

We get:

A = 0011 0000 0

S = 1101 0000 0

P = 0000 1100 0

Perform the loop four times:

1 P = 0000 1100 0. The last two bits are 00.

P = 0000 0110 0. A right shift.

2 P = 0000 0110 0. The last two bits are 00.

P = 0000 0011 0. A right shift.

3 P = 0000 0011 0. The last two bits are 10.

P = 1101 0011 0.

P = P + S.

P = 1110 1001 1. Right shift.

4 P = 1110 1001 1. The last two bits are 11.

P = 1111 0100 1. Right shift.

The final product is 1111 0100, which is -12.

### 3.3 MODIFIED BOOTH'S ALGORITHM

One of the many solutions of realizing high speed multipliers is enhancing parallelism which helps in decreasing the number of subsequent calculation levels. The original version of Booth algorithm (Radix-2) had two particular drawbacks. They were:

- The number of add-subtract operations and shift operations become variable and causes inconvenience in designing parallel multipliers.
- The algorithm becomes inefficient when there are isolated 1's.

These problems are overwhelmed by using modified Radix4 Booth algorithm which scans strings of three bits using the algorithm given below:

1) Lengthen the sign bit 1 position if necessary to ensure that n is even.
2) Add a 0 to the right of the LSB of the multiplier.
3) Corresponding to the value of each vector, each Partial Product will be 0, +M, -M, +2M or -2M.

The negative values of M are made by taking its 2's complement. The multiplication of M is done by shifting M by one bit to the left (in case it's multiplied with 2). Thus, in any case, in designing an n-bit parallel multiplier, only n/2 partial products are generated.

**TABLE 3.1: Modified Booth's Recoding Table**

| ***i+1*** | ***I*** | ***i-1*** | ***add*** |
|---|---|---|---|
| 0 | 0 | 0 | 0*M |
| 0 | 0 | 1 | 1*M |
| 0 | 1 | 0 | 1*M |
| 0 | 1 | 1 | 2*M |
| 1 | 0 | 0 | -2*M |
| 1 | 0 | 1 | -1*M |
| 1 | 1 | 0 | -1*M |
| 1 | 1 | 1 | 0*M |

# CHAPTER 4

## SWITCHING ACTIVITY BASED POWER ESTIMATION

- **DIFFERENT TYPES OF POWER**
- **SAIF FILES**
- **RTL POWER ESTIMATION FLOW**

## 4.1 TYPES OF POWER DISSIPATION

The indispensable figure-of-merit of a digital circuit are speed and power consumption with the spped being measured in terms of a (reciprocal) delay time  or a maximum clock frequency. Efficiency of power could be defined as the total power or also in terms of the switching energy, i.e., the average energy spent for one switching transition of the digital device.

Total power dissipated in a design can be broadly divided in two categories: static and dynamic.

$$P_{tot} = P_{stat} + P_{dyn} = I_{off}V_{DD} + \alpha f_c C_L V_{DD}^2$$

### Static Power

Static power is the power dissipated by a gate when it's not switching. It is caused by the current that flows through the transistors even when they are turned off. From the system's function point of view, static power can be considered as wasted energy as it is not used for any useful purpose. Almost half of design's intake of power may be due to static power at the latest process nodes (65nm) and is growing.

$$P_{stat} = I_{off}V_{DD}$$

### Dynamic Power

Dynamic power is the power dissipated when the circuit is active *i.e.* while performing some function. Dynamic power can be further divided into two components: Switching power and internal power.

$$P_{dyn} = \alpha f_c C_L V_{DD}^2$$

### Switching Power

Switching power can be defined as the power which is dissipated while charging and discharging the output load capacitance. The load capacitance consists of interconnect (net) capacitance and gate capacitances the net is connected to.

The extent of switching power usually depends on the switching activity (which is related to the operating frequency) of the cell. The switching power increases with increase in logic transitions at the cell output

### Internal Power

Internal power is consumed within a cell while charging and discharging internal cell capacitances. Short-circuit power is also included in the Internal power. Both P and N type transistors are on simultaneously during the logic transitions for a brief time resulting to direct connection from $V_{DD}$ rail to ground rail.

## 4.2 SWITCHING ACTIVITY INTERCHANGE FORMAT (SAIF) FILES

As noted above the dynamic power consumed by a circuit depends on the logic transitions which occur within the design while operating. Therefore, for power estimation and optimization we need to accurately specify this information (called switching activity) to the tool performing these tasks. Dynamic power represents the majority of total power. The SAIF (from Synopsys) file stores the switching activity of the design in ASCII format. The SAIF file can then be used to allow switching activity information between the power tools and simulators.

Switching Activity in SAIF file relies on static probability and toggle rate. Following is the definition of Static Probability and Toggle Rate.

**Static Probability**

Static probability is the likelihood that a signal is at a specific logic state; it is expressed as a number between 0 and 1 where SP1 is the static probability that a signal is at logic-1 and SP0 is the static probability that the signal is at logic-0.

You can calculate the static probability as a ratio of the time period for which the signal is at a certain logic state relative to the total simulation time. For example, if SP1 = 0.70, the signal is at logic 1 state 70% of the time. Synopsys power compiler tools use SP1 when modeling switching activity.

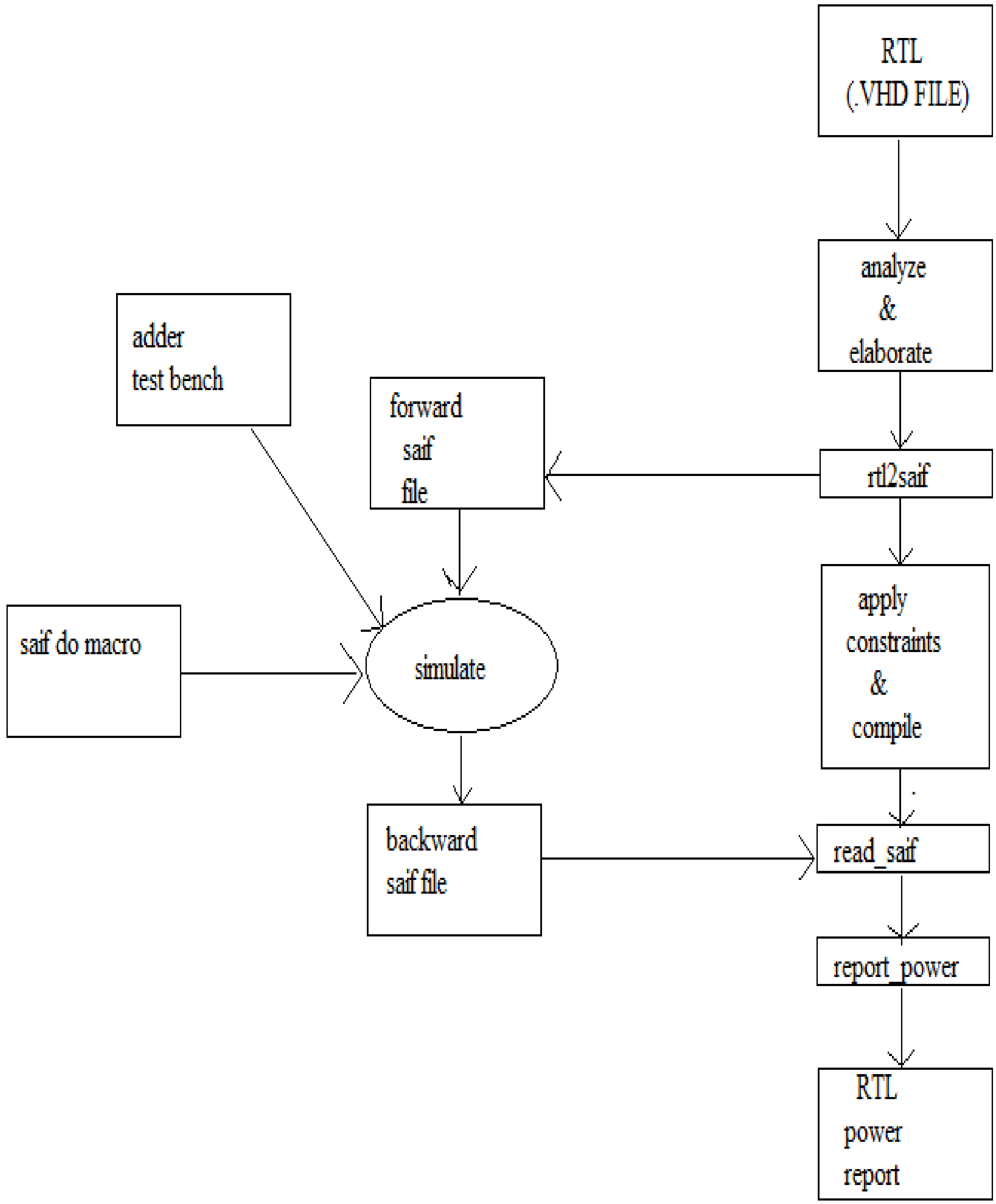
**Toggle Rate**

The toggle rate of a design object is defined as the number of logic-0-to-logic-1 and logic-1-to- logic-0 transitions of the design object, such as a pin, net or port, per unit of time. The toggle rate is written as TR.

## 4.3 RTL POWER ESTIMATION FLOW

This section introduces the RTL Power Estimation Flow i.e. a flow which one can implement to estimate the power consumption of one's design at RTL level using Synopsys Design Compiler and Mentor Graphics Modelsim. Power estimated which is based on the RTL design cannot be said as accurate but it can be used as example to explore different architectures and to evaluate their power consumption. The figure given below illustrates the RTL power estimation flow.

**Figure 4.1: RTL Power Estimation Flow**


RTL
(.VHD FILE)
analyze
&
elaborate
adder
test bench
forward
saif
file
rtl2saif
saif do macro
simulate
apply
constraints
&
compile
backward
saif file
read_saif
report_power
RTL
power
report

**For RTL power estimation one needs:**

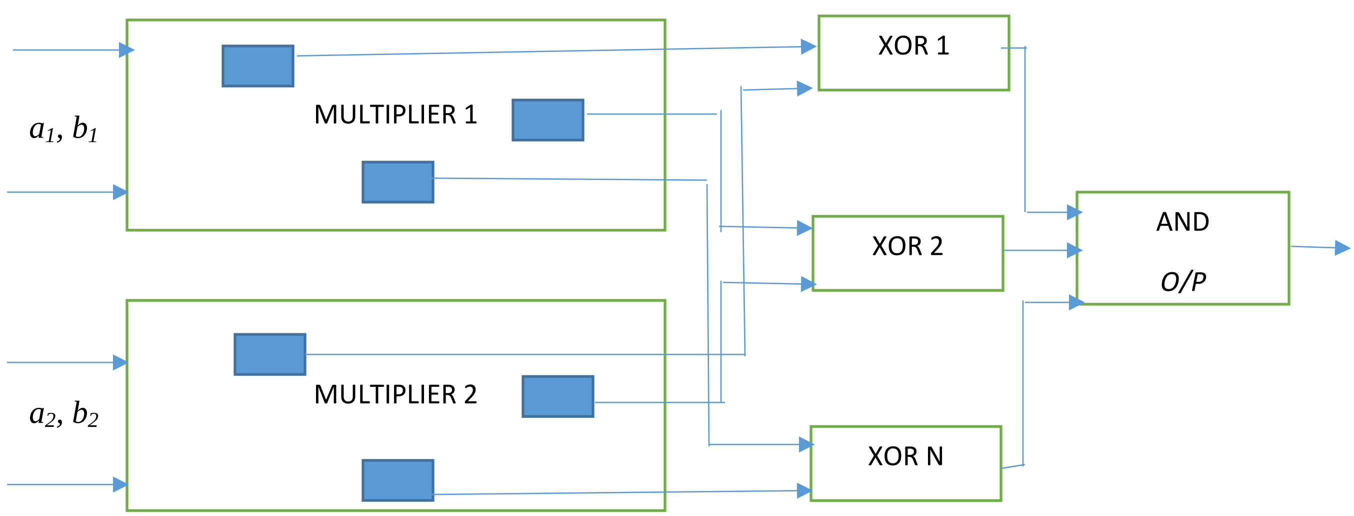
- Synthesizable HDL description of your design (.vhdl file)
- Testbench of the design
- Testbench should generate stimulus for simulation that corresponds to the normal operation of the design
- SAIF forward annotation file (generated by Design Compiler)
    - : - the file provides information to the simulator about the objects in RTL design that should be monitored for switching activity during simulation.
- SAIF do macro file
    - :-the macro file contains ModelSim commands that are needed to record switching activity during simulation
- SAIF backward annotation file (generated by ModelSim simulator)
    - :-the switching activity recorded during simulation

There are specific commands for the generation of both forward and backward SAIF files.

**For A Better Power Estimation**

The problem of maximum/peak power estimation in CMOS circuits is quite essential for analyzing the performance and reliability of circuits at extreme conditions. Here we have tried to find out input vectors that can cause maximum dynamic power dissipation (maximum toggles) in circuits, in other words maximum toggling. A gate level description of the circuit and an initial input vector $I_0$ is given. Let $S_0$ be the initial state of the circuit after assigning $I_0$ to the primary inputs. Now, it is required to find an input vector $I_1$ such that the pairs $I_0$ and $I_1$ applied in sequence lead to high switching activity in the circuit. Most of the approaches in this literature approach the problem of finding input vectors causing maximum switching activity by looking at the gate level description of the circuit. These approaches use some properties of the gate like fan-out, level of the gate in the circuit etc. to decide the order in which the gates need to be processed. In our approach, we have tried to group gates together and look at one group at a time rather than individual gates. This grouping has helped us to obtain better quality solutions. The grouping strategy we decided was to form FFRs (Fanout free regions) in the circuit[3]. The main idea behind this was that for any given FFR, the difficulty of finding an input vector pair which would cause maximum switching among all possible input vector pairs is directly proportional with respect to the number of gates in the FFR. Moreover, this particular input vector pair will cause every line in the circuit to switch state.

**Figure 4.2: For Pattern Generation**




For our pattern generation we used a calculator named as KBDD, which has been developed by a research group at Carnegie Mellon University. This is a BDD (Binary Decision Diagram) generator along with the functionality of generating SOPs (Sum of Products). Here we took two multipliers of the same architecture. Looking at its RTL schematic we found out the potential gates for switching activities and XORed the corresponding gates. After this all the XOR gates were ANDed together to find out the pattern common to the switching of all the logic blocks inside the multiplier. For switching activity analysis the initial logic levels of the design are to be given more importance as they control the activities in the later stages. So for our design we considered the Partial product generation stage only. Also it has been seen that maximum switching occurs in an Adder when firstly the numbers are added and then subtracted i.e. the two's complement to be added with the other input.

1. First write the logic equations of the circuit for all the possible outputs at different logic levels.
2. Rewrite the logic equations using another set of variables.
3. Analyze the RTL schematic for logic blocks having a high potential for switching activity.
4. XOR the corresponding outputs of those logic blocks for the two sets of equation.
5. AND all the XORed outputs to get simultaneous switching activity in the circuit.
6. Generate the SOP of the final equation. Find out the input combinations for which the SOP is satisfied.

# CHAPTER 5

## IMPLEMENTATION AND RESULTS

- **PROGRAMS FOR MULTIPLIERS**
- **OUTPUT WAVEFORMS**
- **RESULTS FROM POWER ANALYSIS**

## 5.1 PROGRAM FOR RADIX-4 MULTIPLIER

### Using Ripple Carry Adder

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity R4MUL_RCA is
    Port (a, b : in STD_LOGIC_VECTOR (31 downto 0);
           mul: inout std_logic_vector(63 downto 0);
         overflow: out std_logic);
end R4MUL_RCA;

architecture Behavioral of R4MUL_RCA is
component RADIX4_ENCODER is
    Port ( x : in  STD_LOGIC_VECTOR (31 downto 0);
           arg : in STD_LOGIC_VECTOR (2 downto 0);
                  pp : inout  STD_LOGIC_VECTOR (63 downto 0));
end component;
component fulladder
    Port (a, b, cin: in STD_LOGIC;
           sum, cout: out STD_LOGIC);
end component;
component RCA64 is
    Port (a, b: in STD_LOGIC_VECTOR (63 downto 0) ;
           add: out STD_LOGIC_VECTOR (63 downto 0);
           cout: out  STD_LOGIC);
end component;
signal arg1, arg2, arg3, arg4: std_logic_vector(2 downto 0);
signal arg5, arg6, arg7, arg8: std_logic_vector(2 downto 0);
```

```
signal arg9, arg10, arg11, arg12: std_logic_vector(2 downto 0);

signal arg13, arg14, arg15, arg16: std_logic_vector(2 downto 0);

signal tt: std_logic_vector(32 downto 0);

signal              s1,s2,s3,s4,s5,s6,s7,s8,s9,s10,s11,s12,s13,s14,s15:
std_logic_vector(63 downto 0);

signal   sum1,sum2,sum3,sum4,sum5,sum6,sum7,sum8:   std_logic_vector(63
downto 0);

signal   sum9,sum10,sum11,sum12,sum13,sum14,sum15:   std_logic_vector(63
downto 0);

signal y: std_logic_vector(15 downto 0);

signal pp1, pp2, pp3, pp4, pp5, pp6, pp7, pp8 : STD_LOGIC_VECTOR (63
downto 0);

signal pp9, pp10, pp11, pp12, pp13, pp14, pp15, pp16: STD_LOGIC_VECTOR
(63 downto 0);

begin

tt<= a(31 downto 0)&'0';

arg1<=tt(2 downto 0);

arg2<=tt(4 downto 2);

arg3<=tt(6 downto 4);

arg4<=tt(8 downto 6);

arg5<=tt(10 downto 8);

arg6<=tt(12 downto 10);

arg7<=tt(14 downto 12);

arg8<=tt(16 downto 14);

arg9<=tt(18 downto 16);

arg10<=tt(20 downto 18);

arg11<=tt(22 downto 20);

arg12<=tt(24 downto 22);

arg13<=tt(26 downto 24);

arg14<=tt(28 downto 26);

arg15<=tt(30 downto 28);

arg16<=tt(32 downto 30);


u1: RADIX4_ENCODER port map(b(31 downto 0), arg1, pp1);
```

```
u2: RADIX4_ENCODER port map(b(31 downto 0), arg2, pp2);
u3: RADIX4_ENCODER port map(b(31 downto 0), arg3, pp3);
u4: RADIX4_ENCODER port map(b(31 downto 0), arg4, pp4);
u5: RADIX4_ENCODER port map(b(31 downto 0), arg5, pp5);
u6: RADIX4_ENCODER port map(b(31 downto 0), arg6, pp6);
u7: RADIX4_ENCODER port map(b(31 downto 0), arg7, pp7);
u8: RADIX4_ENCODER port map(b(31 downto 0), arg8, pp8);
u9: RADIX4_ENCODER port map(b(31 downto 0), arg9, pp9);
u10: RADIX4_ENCODER port map(b(31 downto 0), arg10, pp10);
u11: RADIX4_ENCODER port map(b(31 downto 0), arg11, pp11);
u12: RADIX4_ENCODER port map(b(31 downto 0), arg12, pp12);
u13: RADIX4_ENCODER port map(b(31 downto 0), arg13, pp13);
u14: RADIX4_ENCODER port map(b(31 downto 0), arg14, pp14);
u15: RADIX4_ENCODER port map(b(31 downto 0), arg15, pp15);
u16: RADIX4_ENCODER port map(b(31 downto 0), arg16, pp16);

s1<= pp2(61 downto 0)&"00";
s2<= pp3(59 downto 0)&"0000";
s3<= pp4(57 downto 0)&"000000";
s4<= pp5(55 downto 0)&"00000000";
s5<= pp6(53 downto 0)&"0000000000";
s6<= pp7(51 downto 0)&"000000000000";
s7<= pp8(49 downto 0)&"00000000000000";
s8<= pp9(47 downto 0)&"0000000000000000";
s9<= pp10(45 downto 0)&"000000000000000000";
s10<= pp11(43 downto 0)&"00000000000000000000";
s11<= pp12(41 downto 0)&"0000000000000000000000";
s12<= pp13(39 downto 0)&"000000000000000000000000";
s13<= pp14(37 downto 0)&"00000000000000000000000000";
s14<= pp15(35 downto 0)&"0000000000000000000000000000";
s15<= pp16(33 downto 0)&"000000000000000000000000000000";
```

```
h1: RCA64 port map(pp1, s1, sum1, y(0));
h2: RCA64 port map(sum1, s2, sum2, y(1));
h3: RCA64 port map(sum2, s3, sum3, y(2));
h4: RCA64 port map(sum3, s4, sum4, y(3));
h5: RCA64 port map(sum4, s5, sum5, y(4));
h6: RCA64 port map(sum5, s6, sum6, y(5));
h7: RCA64 port map(sum6, s7, sum7, y(6));
h8: RCA64 port map(sum7, s8, sum8, y(7));
h9: RCA64 port map(sum8, s9, sum9, y(8));
h10: RCA64 port map(sum9, s10, sum10, y(9));
h11: RCA64 port map(sum10, s11, sum11, y(10));
h12: RCA64 port map(sum11, s12, sum12, y(11));
h13: RCA64 port map(sum12, s13, sum13, y(12));
h14: RCA64 port map(sum13, s14, sum14, y(13));
h15: RCA64 port map(sum14, s15, mul, overflow);
end Behavioral;
```

**Using Carry Look-Ahead Adders**

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity r4mulcla is
    Port ( a, b : in  STD_LOGIC_VECTOR (31 downto 0);
           mul: inout std_logic_vector(63 downto 0);
                  overflow: out std_logic);
end r4mulcla;

architecture Behavioral of r4mulcla is
component r4encoder is
    Port ( x : in  STD_LOGIC_VECTOR (31 downto 0);
```

```
          arg : in STD_LOGIC_VECTOR (2 downto 0);
                 pp : inout  STD_LOGIC_VECTOR (63 downto 0));
end component;
component cla is
    Port ( p,g : in  STD_LOGIC_VECTOR (7 downto 0);
          cin: in std_logic;
                 sum : out  STD_LOGIC_VECTOR (7 downto 0));
end component;
component PG_gen is
    Port ( p, g : in  STD_LOGIC_VECTOR (7 downto 0);
          iP, iG : out  STD_LOGIC);
end component;
component carrygen is
    Port ( p, g : in  STD_LOGIC_VECTOR (7 downto 0);
          c1 : in  STD_LOGIC;
          c0 : out  STD_LOGIC_VECTOR (7 downto 0));
end component;
component cla_64bit is
    Port ( a, b : in  STD_LOGIC_VECTOR (63 downto 0);
                 cin: in STD_LOGIC;
          sum : out  STD_LOGIC_VECTOR (63 downto 0);
          cout : out  STD_LOGIC);
end component;
signal arg1, arg2, arg3, arg4: std_logic_vector(2 downto 0);
signal arg5, arg6, arg7, arg8: std_logic_vector(2 downto 0);
signal arg9, arg10, arg11, arg12: std_logic_vector(2 downto 0);
signal arg13, arg14, arg15, arg16: std_logic_vector(2 downto 0);
signal tt: std_logic_vector(32 downto 0);
signal              s1,s2,s3,s4,s5,s6,s7,s8,s9,s10,s11,s12,s13,s14,s15:
std_logic_vector(63 downto 0);
signal   sum1,sum2,sum3,sum4,sum5,sum6,sum7,sum8:   std_logic_vector(63
downto 0);
```

```
signal  sum9,sum10,sum11,sum12,sum13,sum14,sum15:  std_logic_vector(63
downto 0);

signal pp1, pp2, pp3, pp4, pp5, pp6, pp7, pp8 : STD_LOGIC_VECTOR (63
downto 0);

signal pp9, pp10, pp11, pp12, pp13, pp14, pp15, pp16: STD_LOGIC_VECTOR
(63 downto 0);

signal y: std_logic_vector(15 downto 0);

signal c1:std_logic;

begin

c1<='0';

tt<= a(31 downto 0)&'0';

arg1<=tt(2 downto 0);

arg2<=tt(4 downto 2);

arg3<=tt(6 downto 4);

arg4<=tt(8 downto 6);

arg5<=tt(10 downto 8);

arg6<=tt(12 downto 10);

arg7<=tt(14 downto 12);

arg8<=tt(16 downto 14);

arg9<=tt(18 downto 16);

arg10<=tt(20 downto 18);

arg11<=tt(22 downto 20);

arg12<=tt(24 downto 22);

arg13<=tt(26 downto 24);

arg14<=tt(28 downto 26);

arg15<=tt(30 downto 28);

arg16<=tt(32 downto 30);


u1: r4encoder port map(b(31 downto 0), arg1, pp1);

u2: r4encoder port map(b(31 downto 0), arg2, pp2);

u3: r4encoder port map(b(31 downto 0), arg3, pp3);

u4: r4encoder port map(b(31 downto 0), arg4, pp4);

u5: r4encoder port map(b(31 downto 0), arg5, pp5);
```

```
u6: r4encoder port map(b(31 downto 0), arg6, pp6);
u7: r4encoder port map(b(31 downto 0), arg7, pp7);
u8: r4encoder port map(b(31 downto 0), arg8, pp8);
u9: r4encoder port map(b(31 downto 0), arg9, pp9);
u10: r4encoder port map(b(31 downto 0), arg10, pp10);
u11: r4encoder port map(b(31 downto 0), arg11, pp11);
u12: r4encoder port map(b(31 downto 0), arg12, pp12);
u13: r4encoder port map(b(31 downto 0), arg13, pp13);
u14: r4encoder port map(b(31 downto 0), arg14, pp14);
u15: r4encoder port map(b(31 downto 0), arg15, pp15);
u16: r4encoder port map(b(31 downto 0), arg16, pp16);

s1<= pp2(61 downto 0)&"00";
s2<= pp3(59 downto 0)&"0000";
s3<= pp4(57 downto 0)&"000000";
s4<= pp5(55 downto 0)&"00000000";
s5<= pp6(53 downto 0)&"0000000000";
s6<= pp7(51 downto 0)&"000000000000";
s7<= pp8(49 downto 0)&"00000000000000";
s8<= pp9(47 downto 0)&"0000000000000000";
s9<= pp10(45 downto 0)&"000000000000000000";
s10<= pp11(43 downto 0)&"00000000000000000000";
s11<= pp12(41 downto 0)&"0000000000000000000000";
s12<= pp13(39 downto 0)&"000000000000000000000000";
s13<= pp14(37 downto 0)&"00000000000000000000000000";
s14<= pp15(35 downto 0)&"0000000000000000000000000000";
s15<= pp16(33 downto 0)&"000000000000000000000000000000";

h1: cla_64bit port map(pp1, s1, c1, sum1, y(0));
h2: cla_64bit port map(sum1, s2, c1, sum2, y(1));
h3: cla_64bit port map(sum2, s3, c1, sum3, y(2));
h4: cla_64bit port map(sum3, s4, c1, sum4, y(3));
```

```
h5: cla_64bit port map(sum4, s5, c1, sum5, y(4));
h6: cla_64bit port map(sum5, s6, c1, sum6, y(5));
h7: cla_64bit port map(sum6, s7, c1, sum7, y(6));
h8: cla_64bit port map(sum7, s8, c1, sum8, y(7));
h9: cla_64bit port map(sum8, s9, c1, sum9, y(8));
h10: cla_64bit port map(sum9, s10, c1, sum10, y(9));
h11: cla_64bit port map(sum10, s11, c1, sum11, y(10));
h12: cla_64bit port map(sum11, s12, c1, sum12, y(11));
h13: cla_64bit port map(sum12, s13, c1, sum13, y(12));
h14: cla_64bit port map(sum13, s14, c1, sum14, y(13));
h15: cla_64bit port map(sum14, s15, c1, mul, overflow);
end Behavioral;
```

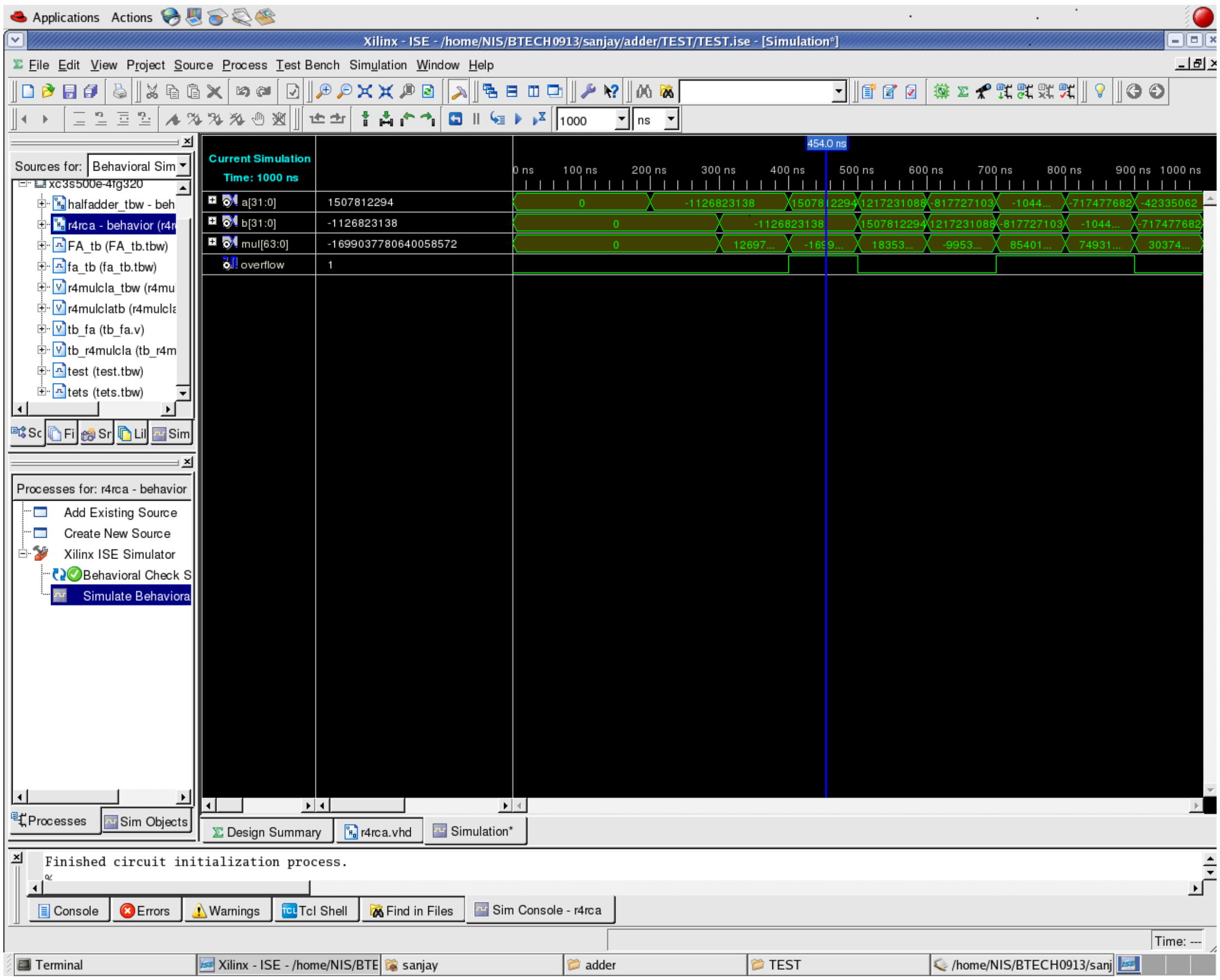
**Modified Booth Encoder**

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity RADIX4_ENCODER is
generic(N: integer:= 32);
    Port ( x : in  STD_LOGIC_VECTOR (N-1 downto 0);
           arg : in STD_LOGIC_VECTOR (2 downto 0);
                  pp : inout  STD_LOGIC_VECTOR (2*N-1 downto 0));
end RADIX4_ENCODER;

architecture Behavioral of RADIX4_ENCODER is
begin
process(arg, x)
variable temp, temp1, temp2: std_logic_vector(N downto 0);
begin
```

```
if x(N-1)='1' then
temp:= '1'&x(N-1 downto 0);
else
temp:= '0'&x(N-1 downto 0);
end if;
if(arg="001"or arg="010") then
temp1:= temp;
elsif(arg="101" or arg="110") then
temp1:= not(temp) + "000000001";
elsif(arg="011") then
temp1:= temp(N-1 downto 0)&'0';
elsif(arg="100") then
temp2:= not(temp) + "000000001";
temp1:= temp2(N-1 downto 0)&'0';
else
temp1:= (others=>'0');
end if;
pp<= sxt(temp1, 2*N);
end process;
end Behavioral;
```

## 5.2 OUTPUT WAVEFORMS

### Testbench Waveform generation using XIlinx




### Scirocco & VirSim Logic Simulation

The commands executed in the Terminal window for the simulation of the Radix-4 multiplier using RCA:

// VHDL analysis of different components and the multiplier program

*vhdlan fulladder.vhd*

*vhdlan RCA64.vhd*

*vhdlan RADIX4_ENCODER.vhd*

*vhdlan R4_MUL_RCA.vhd*

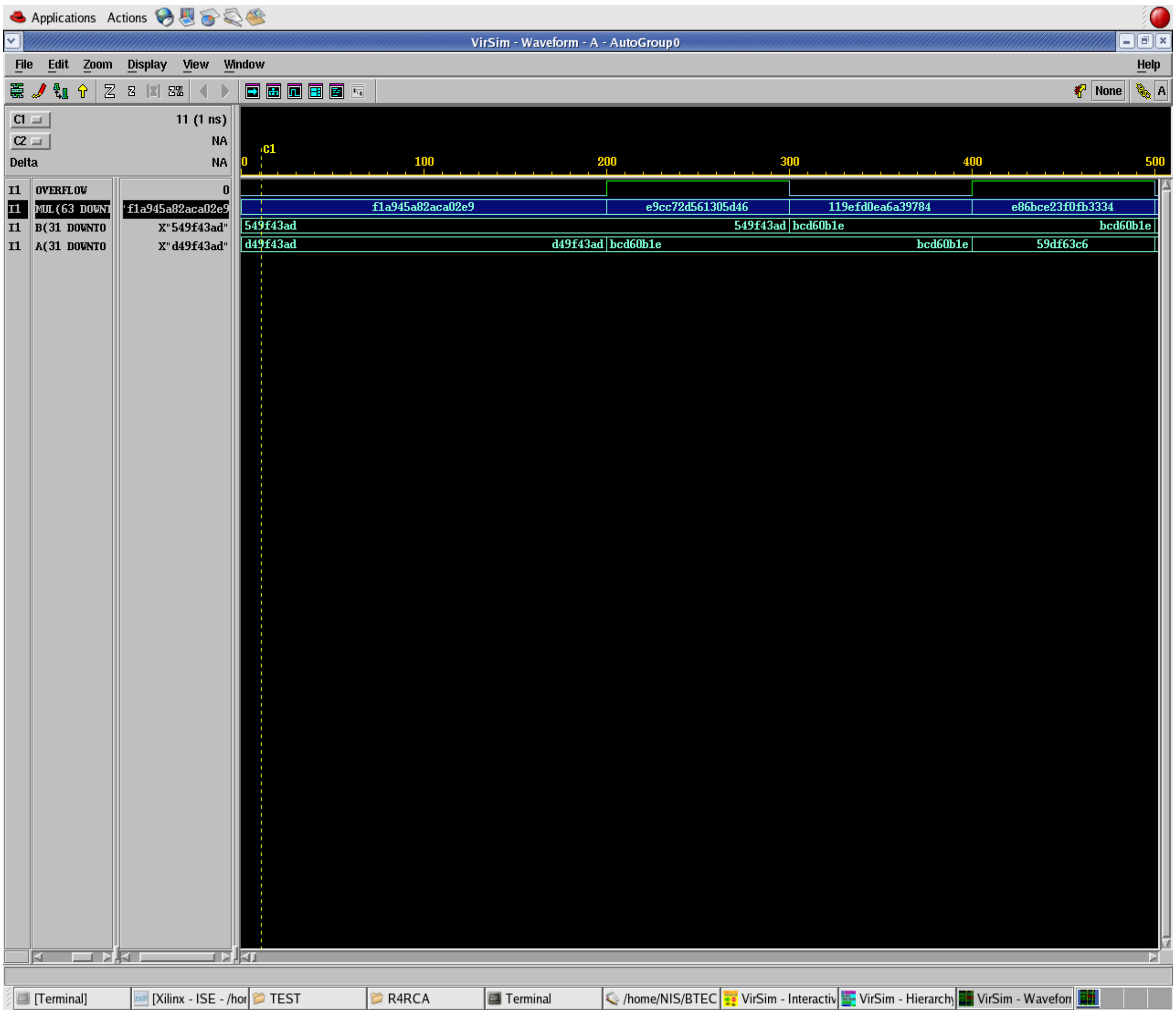
*// Verilog analysis of the test bench*

*vlogan test_r4rca.v*

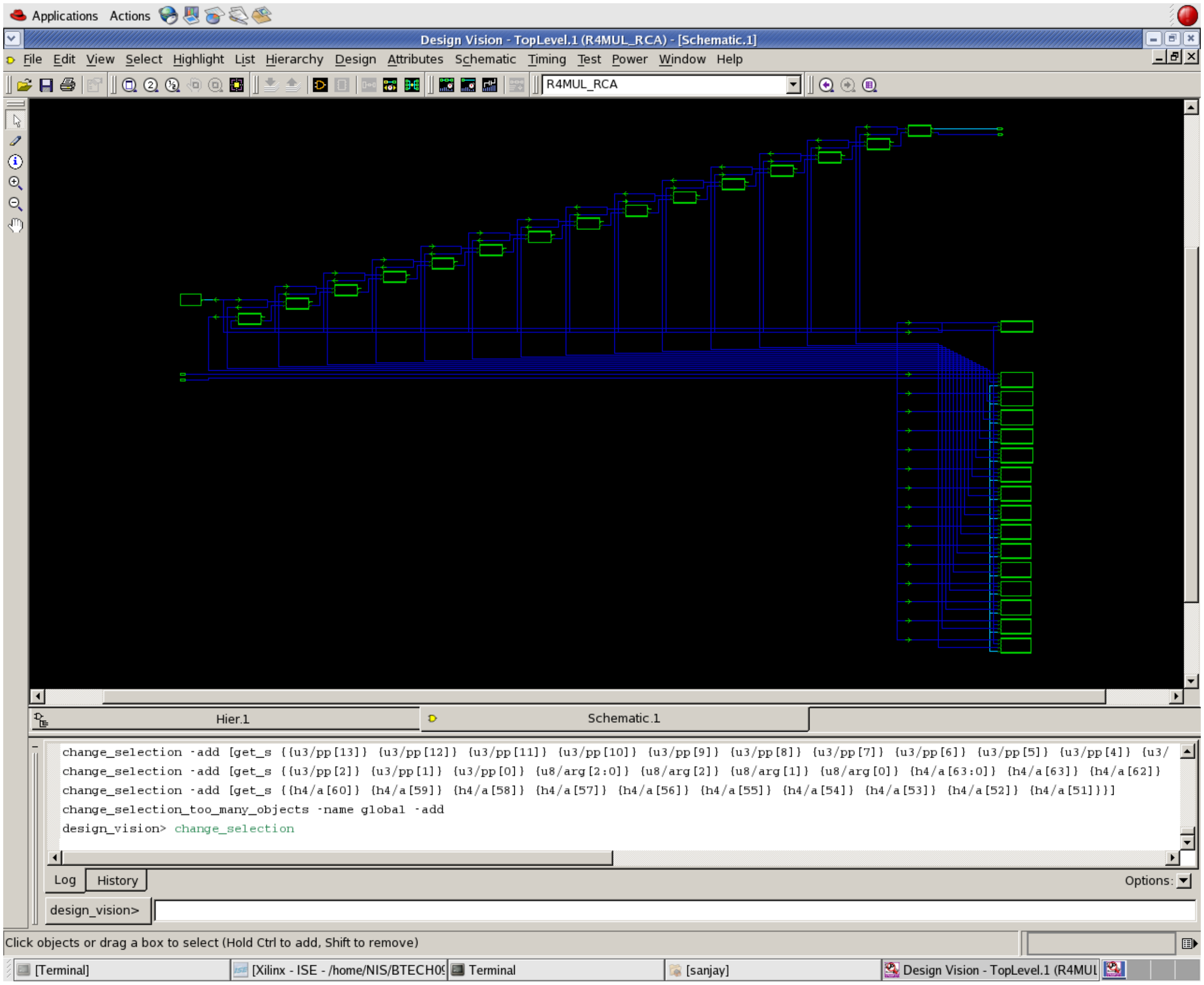
*// Analysis of the Configuration statement*

*scs CFG*

*scirocco &*



## Analysis & Elaboration of the design using Design Vision



## 5.3 RESULTS FROM POWER ANALYSIS

Estimation of average power (P1) consumed by the Booth multiplier using RCA & CLA architecture

**TABLE 5.1: Average Power Estimation for Different Multipliers**

| *Name of design* | *Cell Internal Power (in mW)* | *Net Switching Power (in mW)* | *Total dynamic Power (P1) (in mW)* | *Cell Leakage Power (in uW)* | *Area (in $um^2$)* |
|---|---|---|---|---|---|
| Radix-4 RCA multiplier | 4.5215 (71%) | 1.8224 (29%) | 6.3439 (100%) | 93.0186 | 19850.040021 |
| Radix-4 CLA multiplier | 4.9955 (68%) | 2.3702 (32%) | 7.3658 (100%) | 108.5598 | 24413.039974 |

After the estimation of above data we carried out further power analysis on the design using RCA architecture since it consumes less power and less area.

***Commands Used For SAIF File Generation***

***For Forward Saif File:***

*set power_preserve_rtl_hier_names true*

*# Analyze the design*

*analyze -format vhdl -library WORK R4_MUL_RCA.vhd*

# Elaboration

*elaborate R4_MUL_RCA –architecture behavioral -library WORK*

*# Generates forward saif file*

*rtl2saif -output r4mul_rca_fw.saif -design R4_MUL_RCA*

*write -hierarchy -format ddc -output r4mul_rca_elaborated.ddc*

***For Backward Saif File:***

*# Analyze the multiplier*

*vhdlan R4_MUL_RCA.vhd*

*# Analyze full adder*

*vhdlan fulladder.vhd*

*# Analyze Booth encoder*

*vhdlan RADIX4_ENCODER.vhd*

*# Analyze RCA 64-bit adder*

*vhdlan RCA64.vhd*

*# Analyze the testbench*

*vlogan test_r4rca.v*

*vcs –debug_all –mhdl test_r4rca.v*

*# Generate backward saif file*

*./simv*

***Creating Power Report:***

*# Read the elaborated design (or re-elaborate the design)*

*read_file -format ddc {r4mul_rca_elaborated.ddc}*

*# Compile the design*

*Compile*

*# Read backward annotation SAIF file*

*read_saif -input r4mul_rca_bw.saif -instance_name R4_MUL_RCA*

*# Run power reporting command*

*report_power*

**TABLE 5.2: Power Estimation from SAIF files**

| Power | For input combination-1 | For input combination-2 |
|---|---|---|
| Cell Internal Power | 306.8672 uW (73%) | 89.5134 uW (78%) |
| Net Switching Power | 116.2471 uW (27%) | 25.5467 uW (22%) |
| Total Dynamic Power | 423.1143 uW (100%) | 115.0601 uW (100%) |
| Cell Leakage Power | 92.7883 uW | 90.9827 uW |

# CHAPTER 6

## CONCLUSION & FUTURE WORK

## REFERENCES

## CONCLUSION AND FUTURE WORK

After going through all the hard work and after facing a lot of problems, we managed to complete the objectives of the project that are to implement Booth's Algorithm for the design of a binary multiplier using different adder architectures and carry out power analysis at various levels.. We analyzed the area occupied and the time delay consumed by different adders and found out an appropriate relationship among the time and area complexity the adders which we have taken into consideration. After comparing all we came to a conclusion that Ripple Carry Adders are best suited for Low Power Applications. Then we turned our focus into the design of Multipliers. First of all we designed a Booth's Radix-2 multiplier and estimated its delay, area and power. Then a radix-4 multiplier was designed. A comparison was done between Radix-2 and Radix-4 algorithm. Comparing data between Radix-2 and Radix-4 booth multipliers we found out that radix-4 consumes less power than radix-2, because radix-4 uses almost a half number of iterations than radix-2 As radix-4 seemed more suitable for the design we carried out further analysis on radix-4 multiplier by using different adder architectures like RCA and CLA. Then we turned our focus into the switching activity based power analysis of the Radix-4 Booth multiplier, and its power estimation. We did power estimation at RTL level using Synopsys Design Compiler.

Further work can be carried out on this project in the power estimation section. Power can be estimated at the gate-level by generating gate-level netlist and also the post layout analysis can be done for this design. Another possible direction can be pursued for higher radix encoding.

## REFERENCES


**[1]** ***A. D. Booth*** *"A signed binary multiplication technique," Quart. J. Mech. Appl. Math., vol.4*

**[2]** *"Design, Analysis and Switching Activity based Power Estimation of Booth Multiplier using Different adder techniques" by* **Arun Kumar P.S, J K Das, Sudeendra Kumar, K K Mahapatra**.

**[3]** *"Controllability-driven power virus generation for digital circuits" by K.Najeeb, Karthik Gururaj, V.Kamakoti.*

**[4]** *Power Compiler Manuals, www.synopsys.com*